# Hardware architecture specification and constraint-based WCET computation

Hajer Herbegue, Hugues Cassé, Mamoun Filali, Christine Rochange
Institut de Recherche en Informatique de Toulouse
CNRS Université de Toulouse
firstname.lastname@irit.fr

***Abstract*—The analysis of the worst-case execution times is necessary in the design of critical real-time systems. To get sound and precise times, the WCET analysis for these systems must be performed on binary code and based on static analysis. OTAWA, a tool providing WCET computation, uses the Sim-nML language to describe the instruction set and XML files to describe the micro-architecture. The latter information is usually inadequate to describe real architectures and, therefore, requires specific modifications, currently performed by hand, to allow correct time calculation. In this paper, we propose to extend Sim-nML in order to support the description of modern microarchitecture features along the instruction set description and to seamlessly derive the time calculation. This time computation is specified as a constraint solving problem that is automatically synthesized from the extended Sim-nML. Thanks to its declarative aspect, this approach makes easier and modular the description of complex features of microprocessors while maintaining a sound process to compute times.**

## I. Introduction

The main characteristic of hard real-time systems is that they must guarantee a correct timing behavior. A hard real-time task has a deadline to meet, otherwise the real-time system may fail with catastrophic consequences. In the design of hard real-time systems, it is absolutely required to know the upper bound of the execution time of each task, named the worst-case execution time (WCET), in order to determine a task schedule that ensures the deadlines to be met. Different methods are applied to compute the WCET [30] of critical tasks, such as simulation or static analysis of the executable. Estimating the WCET using simulation supposes that we have all possible combinations of input data values that cover all the possible execution paths and of the initial system states, which is not possible in general. Static analysis is a safe method that can estimate the actual WCET with respect to the target architecture. Generally, the WCET computation is based on the analysis of the program control flow. Feasible and infeasible paths are identified, as well as (program) flow facts and loop bounds. Each execution path of the analyzed program is split into code snippets, called *basic blocks*. The WCET of a program is expressed as the sum of the execution times of the basic blocks that compose the longest execution path, weighted by their respective execution counts. To search the longest path, the IPET (Implicit Path Enumeration Technique) [18] is the most used and successful. It is based on the Integer Linear Program [29] to formulate the problem. The execution time of basic blocks are estimated by simulation, abstract interpretation or execution graphs.

In the present work, we consider the OTAWA framework as a starting point. OTAWA is dedicated to the development of WCET analyzers. It includes a number of analysis tools that are based on Architecture Description Languages (ADL) to specify architectures. The Sim-nML language is used to describe the instruction set architecture and the basic block execution time is computed by execution graphs [24] [17]. Our long-term goal is to have a fully certified WCET computation chain. We present an ADL-based approach [21], from which we derive a constraint-based method for WCET computation. In order to do so, we extend the Sim-nML language to describe the hardware structure of the processor and the execution model of instructions. We superimpose the effective use of resources on the basic description of instructions. This approach aims at estimating the execution time of a basic block and enabling the verification of the architecture specifications.

The paper is organized as follows. Section II gives an overview of the OTAWA framework, its limitations and our contribution. Section III presents the extension of the Sim-nML language. In Section IV, we introduce a constraint-based method for WCET computation. We experiment our contribution on the ARMv5 instruction set with an out-of-order superscalar processor. The implementation and experimental results are presented in Section V. Related work is overviewed in Section VI. Section VII concludes the paper.

## II. WCET ANALYSIS OVERVIEW

### A. OTAWA framework

OTAWA [4] is a framework dedicated to WCET computation that is able to integrate different kinds of methods of computation. The framework OTAWA is founded on an abstraction layer that describes the target hardware and the instruction set architecture (ISA), as well as the binary code under analysis. The ISA specification is expressed in the Sim-nML language. The hardware configuration is specified in XML files that describe the pipeline structure, the caches and the memory. Yet only some architectures can be handled by such a model. Pipelines described at that level contain as many stages as required, among 4 stage models: fetch, lazy, execute and commit. The main goal of such a description is to represent where time is spent and to express the capacities of hardware resources.

The framework includes a flow analyzer that builds the control flow graph (CFG) of the program from the binary code. Flow facts (e.g. loop bounds, instruction cache analyzer output) are retrieved from annotated source code and then added to the CFG. The architecture abstraction, the program representation and the flow facts are the input of different analyses performed by OTAWA. A pipeline analysis consists in building execution graphs to model the execution of basic blocks on the pipeline and then compute the corresponding times [24].

### B. Motivation and methodology

This work is not intended to rebuild the OTAWA framework, but to reuse some of its modules in our approach. Our analysis method is fully automatic and aims at computing the WCET of a basic block with respect to its execution context. Actually, the execution time of a basic block depends on the instructions executed before it. In our analysis, we consider a basic block with the sequence of blocks executed before it that directly affect the timing of the block, referred to as the *prefix*. We call *instruction path*, the instruction sequence consisting in the prefix followed by the basic block.
We think it is necessary to enhance the expressivity of OTAWA with regard to the processor description language. In fact, the time analysis relies on both the ISA description and the hardware architecture description, which are structured into two modules, using different formalisms. The currently used model to describe the processor is too restrictive and does not allow to handle features often found in the microprocessors (instructions with exotics execution path in the pipeline, instructions decomposed in several micro-instructions, instructions with a variable duration in the functional unit, etc) and, as a result, cannot be fully automated. Actually, these features are implemented by hand using OTAWA libraries and integrated to the analysis during the build of the execution graph. The aim of our work has been to dissociate the hardware processor description (structure and features) and the subsequent analyses. In order to do so, we have enhanced the Sim-nML language to take into account the general hardware description of complex processors and the execution model of the instruction set. The presented analysis is generic, based on the architecture description and the program. Moreover, for our analysis, we have tried to reuse well known constraint specification languages [27] and resolution tools [2], [3], in order to apply existing efficient constraint solvers. Actually, our constraint specification is build on the Global Constrait Catalog [5] and does not embed any specific constraint resolution method. As we will see in section IV, since our constraints are generic, we can also reuse any constraint solver. Figure 1[1] describes the new work flow to

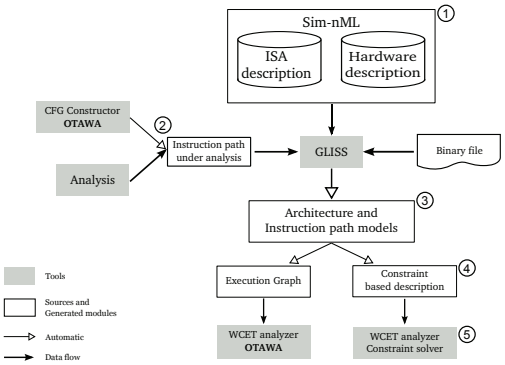


Figure 1: Work flow for ADL-based WCET estimation.

compute the worst case execution time of a basic block. The user provides a Sim-nML architecture description while the CFG constructor performs the path analysis on the program binary and provides the instruction path to analyze. From the architecture description and the instruction path, we generate an abstract description for the architecture and for the instruction path. This is achieved using the GLISS tool [23] which includes a Sim-nML parser and a C-library generator. We have extended GLISS to take into account the extension of Sim-nML language and to automatically generate the architecture and the instruction path models. From the architecture and the instruction path models, we generate an execution graph, that can be analyzed by the OTAWA framework. We also have implemented a module that generates a constraint-based description, that can be analyzed with any constraint-based resolution method.

[1]We will use the numbers in the figure for the illustrating example of section V

## III. The Sim-nML language and the proposed extension

An architecture description consists in the description of its hardware components and the supported instruction set. In this section, we review how an instruction set and a hardware architecture are described and how we have extended it.

### A. Instruction set description

Sim-nML [9], [11], [22] is a hierarchical and a highly structured architecture description language used to describe instruction set architecture and to generate processor specific tools such as assemblers, disassemblers, processor simulators, functional simulators, etc. In Sim-nML, the processor model is described as a hierarchical structure using an attributed grammar. Two kinds of rules are used to produce instructions specifications. OR-rules gives alternatives of instruction parts and so represents non-terminal symbols. AND-rules gives the composition of an instruction parts and so defines terminal symbols. AND/OR rules are also used to describe the addressing modes, used to define instruction operands. Instructions and modes are described by pre-defined attributes :

- **syntax**: the textual representation of the instruction or addressing mode in the assembly language.
- **image** : the binary representation of the instruction or the addressing mode in the executable code.
- **action** : the semantics of the instructions.

An example of instruction specification is given in listing 1. The top-level operation, named `instruction`, is the root of all instructions and is given by an AND-rule with a parameter referencing to the `all_instr` rule. This parameter represents any of the instructions supported by the processor. This property is specified using an OR-rule, `all_instr`, listing all the possible family of instructions. The processor instructions presented in the example are `add`, `load` and `branch`. The processor uses two addressing modes, `reg_index` and `mem_index`, representing respectively, register access and memory access modes. Types and temporary variables can be declared to complete instructions and modes definition.

Listing 1: Sim-nML instruction set description

```
// types used in the description
type index = card (2) // register index type
//and-rule : top-level instruction definition
op instruction ( x : all_instr )
  syntax = format ( "%s", x.syntax )
  image = format ( "%8b", x.image )
  action = {PC = PC + 2;
            x.action; }

//or-rules
op all_instr = add | load

// mode definition
mode reg_index (i:index) = R[i]
  syntax = format ("r%d",i)
  image = format("%2b",i)

mode mem_index (i:index) = M[R[i]]
  syntax = format ("(r%d)",i)
  image = format("%2b",i)

// instruction definitions
op add (d:reg_index, src1:reg_index, src2:reg_index)
  syntax = format ("add %s %s %s",dest.syntax, src1.
      syntax, src2.syntax)
  image = format ("00%s%s%s",dest.image, src1.image,
      src2.image)
  action = {dest = src1 + src2 ;}

op load (dest:reg_index, src:mem_index)
  syntax = format ("load %s %s ",dest.syntax, src.
      syntax)
  image = format ("0100%s%s",dest.imag, src.image)
  action = {dest = src;}
```

Resources used in the instruction definitions, like the register `PC`, register bank `R` and memory `M` are part of the hardware architecture description, presented in the next section. [9] details the Sim-nML formalism.

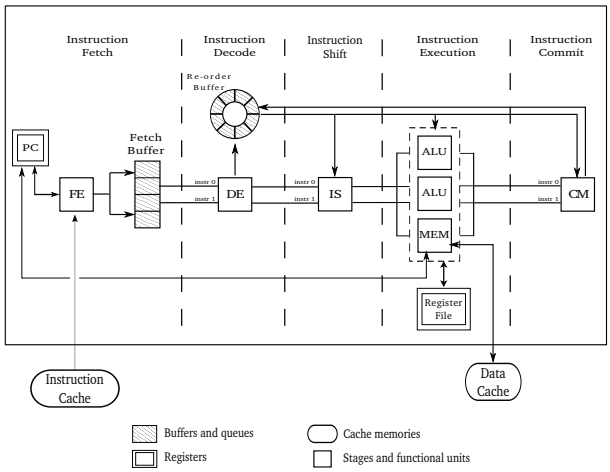


Figure 2: An out-of-order Superscalar Processor Model

### B. Hardware components description

In this section, we present the main features of Sim-nML extension that we rely upon for the WCET computation. To illustrate subsequent explanations, we consider the following example of microprocessor.

***An out-of-order Superscalar Processor:*** Superscalar architectures allow parallel instruction execution. To improve efficiency, modern processors execute also instructions out-of-order: the execution order is governed by the availability of resources rather than the

program order. Figure 2 shows such a processor supporting 2 instructions per cycle. The pipeline includes a fetch stage, a decode stage, an instruction selector stage, an execution stage and a commit stage. The execution stage has three functional units (two arithmetic and logical units and one memory access unit). Only the execution stage supports out-of-order execution to improve usage of its functional units. Fetched instructions are stored in a 4-entry fetch buffer and are pulled by the decode stage. Then, this stage stores them in the 8-entry re-order buffer. This buffer is used to maintain the program order for the commit stage. The instructions execution in the functional units is possibly out-of-order, with respect to the availability of the resources and the data dependencies.

Listing 2: Hardware architecture description

```
stage FE , DE , IS , ALU[2] , MEM , CM

extend FE , DE , IS , CM
 capacity = 2    // degree of super-scalarity
 inorder = true // in-order stages

extend ALU , MEM
 inorder = false // out-of-order stages

extend FE
 blocked = 16 // cache block size in byte

// Fetch Buffer and Re-order Buffer
buffer FBuf [4] , RoB [8]

// registers and memory declaration
reg PC [1, card(32)]     // program counter register
     of 32 bits
reg R [32, card(32)]     // a register bank with 32
     registers of 32 bits
mem M [32, card(8)]      // a main memory with 2^32
     words of 8 bits
```

***Stages:*** Functional units and stages in a pipeline are considered as execution resources, both declared as `stage`. Each stage or functional unit completes a part of an instruction. An instruction is executed at most by one stage or one functional unit, at a time. In the processor description, no limitations on pipeline depth are imposed. In listing 2, one occurrence of fetch stage `FE`, decode stage `DE` and instruction selector stage `IS` are declared. `ALU [2]` means that the processor contains two occurrences of that functional unit.

Our formalism allows a simple and explicit description of the pipeline features, using attributes extending the stages specification. We express superscalarity of a stage with an integer attribute (`capacity`) assigned to stages or functional units. We also assign an attribute `inorder` to stages specification to identify out-of-order executing stages. User can specify other architecture characteristics that are pertinent for subsequent architecture analysis. For example, the cache block size is a relevant parameter to estimate time penalties when instructions are fetched from cache: two instructions belonging to two different cache blocks cannot be fetched simultaneously.

***Buffers:*** Buffers are special hardware components that store instructions temporarily, while awaiting for being processed by some stage, or for some unavailable resources. Buffers and queues slots cannot be read or written by instructions. Also, they are not hardware units executing instructions, but instructions containers. An instruction occupies only one slot of the same buffer type. The key word `buffer` introduces buffers. For example, `FBuf[4]` means that the pipeline contains a 4-entry fetch buffer. We consider each buffer slot as a resource.

***Registers and memories:*** Registers and memories are part of the original Sim-nML language. They are used in instructions and modes definitions. A memory is introduced with key word `mem` and is characterized by the number of bits needed to address it and the size of its elements. The key word `reg` introduces registers. A register bank is given by the data type and the number of registers it contains. Each element of register bank declared with key word `reg`. Memory is considered as a monolithic resource.

### C. Instruction execution model

The instructions own resources and perform actions all along their execution. From the processor point of view, the pipeline executes in its different stages several instructions that access concurrently the available resources. To be executed on a stage, an instruction has to wait for its requirements to be ready (released by previous instructions). Therefore, the execution time of an instruction is impacted by the resources state. So, we extend the instruction set description by an execution model that defines, in a timed sequence, the resources used by an instruction in each step of its execution. This is represented by the attribute `uses` made of a sequence of *clauses* separated by commas. Each clause represents a step in the instruction execution in the pipeline as a list of required resources.

Listing 3: General instruction description

```
op instruction ( x : all_instr)
  syntax = x.syntax
  image = x.image
  action = x.action
  uses = clause_1 , clause_2 , clause_3 , .....
```

***Basic resource request:*** Resource requests can be expressed according to different schemes: non-deterministic and deterministic resource requests. We have a non-deterministic resource request when an instruction requires a resource, which exists in many copies in the architecture, without specifying any occurrence of it. This is specified in the `uses` by giving only the resource identifier. For example, buffers are required in a non-deterministic schema. In fact, when

an instruction requires any slot resource available in the buffer can be assigned. In the processor example presented above, the two ALU units can execute the add instruction (listing 4) while the multiply instruction is only supported by the first occurrence, ALU[0]. This is a deterministic resource access. If the stages may be accessed deterministically or not, the accesses to registers is always deterministic: the register number is specified in the encoding of the instructions. The memory, represented as a monolithic resource, is always deterministically accessed. For the latter resources, it is also required to know if a read or a write access, keywords read and write, is performed.

***Parallel resource request:*** The simultaneous access to many resources is expressed using the '&' operator. An instruction begins execution on a stage when all the required resources of this stage are available. Otherwise, the instruction stalls until its resources are available. In listing 4, to achieve execution on ALU stage, the mul instruction waits until both ALU[0] stage and the registers R[dest], R[src1] and R[src2] are available. If one of these resources is not available, the others will not be allocated until all resources are available. The instruction releases simultaneously acquired resources before it allocates resources of the next step.
In a parallel resource request, no more than one resource of the same type requested in a non-deterministic way is allowed. Actually, an instruction cannot be stored in two buffer slots simultaneously. An instruction cannot be executed, at the same time, by two stages or functional units.

Listing 4: Uses attribute

```
op mul (dest:reg_index,src1:reg_index,src2:reg_index)
  uses = FE & FBuf , DE , IS & RoB ,
         ALU[0] & R[src1].read & R[rsc2].read
         & R[dest].write & RoB , CM

op load (dest : reg_index, src : mem_index)
uses = FE & FBuf :increment_pc #{2}, DE , IS & RoB ,
       MEM & R[dest].write & R[src].read & M.read
       & RoB#{10} :action , CM
```

***Attributed resource request:*** Instructions acquire resources for a limited number of time units. Some actions can be performed when resources are granted or/and when released. A time-action attribute can be specified for clauses. The time attribute defines the duration for which resources are allocated by the instruction. In the specification of the load instruction (listing 4), the fetch stage FE and the FBuf buffer slot are allocated for 2 time units, the decode stage DE for one time unit (default value). If an instruction is stalled, the specified time value does not include the additional time due to the stall. The Sim-nML extension allows specifying an action to perform when resources are acquired and an action to perform when resources are released. For example, we can specify the predefined action attribute to be performed after instruction execution on ALU. The action increment_pc, that increments the program counter, is performed on fetch stage before the instruction cache access.

### *D. Resources and stages allocation strategy*

In a pipeline architecture, instructions are executed in parallel and access concurrently resources. To ensure a deadlock-free instructions execution, a strategy for resource allocation and releasing have to be fixed. Further, the instructions use hardware components (stages, buffers, registers, etc.) differently, since each component has a particular role. We present below the more relevant properties of the proposed resource usage model.

***Resources allocation:*** To move from the current execution step to the next step, an instruction releases the resources that are not required on the next stage. If a resource that the instruction already has is required on the next step, then the resource is not released. If we consider the uses attribute in listing 4, to move forward from the fetch to the decode stage, if the stage DE is available, both resources FE and FBuf, not used on the decode step, are released. However, if the stage DE is not available, the instruction is stalled. In that case, all the resources previously granted by the instruction are released, except the buffer resource (see later). When instruction goes from the IS stage to the ALU unit, the buffer RoB is not released. First acquired by instruction on the IS stage, the RoB buffer is not released until instruction is executed by the commit stage.

***Buffers usage model:*** Buffers are special resources that contain instructions. The fetch buffer, for example, stores instructions since they are fetched from memory cache until the decoding starts. After an instruction is fetched and stored in a buffer slot, if the decode stage is not available, then the instruction stalls on the fetch buffer. So that, the fetch stage is not locked and can continue to fetch instructions from memory. Then, unless the resource is used on the next step, the current buffer slot remains allocated, as long as the instruction is stalled. The only situation where the fetch stage is locked is when an instruction is fetched and there is no free slot in the fetch buffer to allocate. So the fetch stage is locked by that instruction, until a fetch buffer slot becomes available.

### *E. Overview of the internal description language*

This abstract syntax concerns the architecture description and the resource request superposed to Sim-nML clauses. The abstract architecture description is automatically generated from the Sim-nML description. With respect to Sim-nML, the relevant aspects of a

processor description is the list of stages together with the list of unit resources, which are buffers, registers and memories. The abstract representation of clauses (listing 5) is based on the OCCAM language [6], [15].

Listing 5: Data type of clauses

```
type access = Read | Write
type ('a,'s,'r ) clause =
        | USES of ('s,'r ) request * access option
        | PAR  of ('a,'s,'r ) clause list
        | SEQ  of ('a,'s,'r ) clause list
        | ATTR of ('a,'s,'r ) clause * 'a
```

Intuitively, `'a` is the attribute superimposed to clauses, `'s` is the stage type and `'r` is the resource type.

## IV. Constraint-based WCET analysis

### A. The execution time of a basic block

In the most used approach, IPET (Implicit Path Enumeration Technique), the WCET of a program is the maximized objective function of an ILP (Integer Linear programming) problem [18]:

$$maximize \sum_{b \in \mathcal{B}} N_b * C_b$$

where $\mathcal{B}$ is the set of the basic blocks of the program. $C_b$ and $N_b$ denote an estimation of the time and the execution count of a basic block $b$. Since the execution time of a basic block depends on the instructions executed before it, the preceding instructions, called a *prefix*, are involved and form, with the considered basic block, an *instruction path*. The execution cost of a basic block, $c_b$, is defined as the time between the completion of the last instruction in the prefix and the completion of its last instruction[2]. According to the paths of the CFG, different prefixes can be used as the execution context of a basic block. The worst-case execution time of the block is the maximum of the times obtained for every possible prefixes.

Our method aims at deriving a WCET estimate for each basic block. Each instruction is broken down into steps representing the execution of the instruction in a pipeline stage. Therefore, $c_b$ is defined as the time between the last stage of the last instruction of the prefix $i_{p,d}$ and the last stage of the last instruction of the basic block $i_{n,d}$. This is shown in figure 3, where $i_{x,y}.end$ denotes the end date of the step $i_{x,y}$. One can observe that our implementation allows instructions may have different lengths. The cost of the basic block is defined as:

$$C_b = i_{n,d}.end - i_{p,d}.end$$

To evaluate the WCET of the basic block, we describe the execution of an instruction path on a processor as a

[2]This definition makes sense when instructions terminate in program order, which is the case for most microprocessors

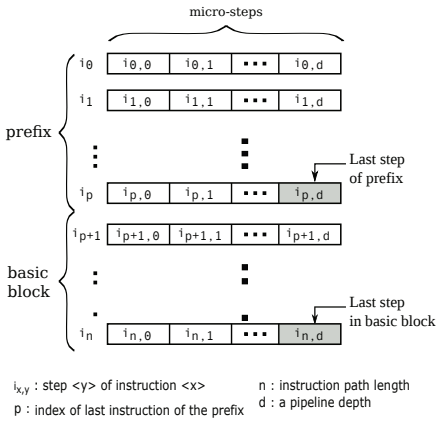


Figure 3: Composition of an instruction path.

Constraint Satisfaction Problem (CSP). Instructions and steps are described with temporal intervals that represent their lifetimes. We express instructions dependencies through constraints intervals. Solving the CSP consists in finding all the possible values of the interval bounds of every step $i_{x,y}$ where the WCET is the minimization of the maximum value of all the $i_{x,y}.end$.

### B. The constraint sub-language

Our goal is to propose a constraint-based declarative approach to compute cost of basic blocks. In the following subsections, we recall the notons that we have used in our proposal.

*1)* ***Allen intervals****:* Allen intervals are used to enforce the relations between the non atomic activities we consider. The attached algebra defines temporal relations between time intervals. 13 (exclusive) basic relations are defined (figure 4) where only 7 relations are depicted (other ones are obtained by symmetry , $=$ being its own symmetrical).

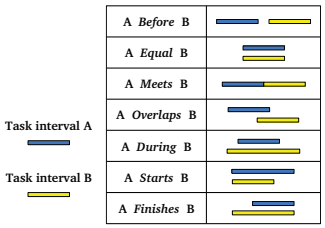


Figure 4: Allen intervals

*2)* ***The StartsBeforeBegin constraint****:* The StartsBeforeBegin constraint is considered to handle effi-

ciently priorities. Given two tasks $i$ and $j$, the StartsBeforeBegin constraint expresses that the task $j$ starts at the same time or after the begining of the task $i$.

*3) **The cumulative constraint**:* The cumulative constraint is used to enforce the concurrency relations, e.g., sharing of resources. With respect to a given resource with $C$ units available, given a list of tasks[3] $\tau = [(i_0, c_0); \ldots; (i_{n-1}, c_{n-1})]$ defined by their respective execution interval $i_i$ and capacity requirement $c_i$, the cumulative constraint Cumulative$(C, \tau)$ expresses that, at any time, the resources granted (exclusively) do not exceed the number $C$ of available resources.

$$\forall t \in [\min_{0 \leq k < n} \{i_k.m\} \ , \ \max_{0 \leq k < n} \{i_k.M\}]. \ \Sigma_k \ c_k \leq C$$

with $i_k.m$ and $i_k.M$ denoting the lower and upper bounds of the interval $i_k$. A particular case of the cumulative arises when the number of available resources is 1: a mutual exclusion constraint. Such a constraint is usually called `disjunctive`. In the following, we apply the Allen constraints to the computation of the basic block cost.

### C. The constraints space

With respect to our concerns, an interval is used to model the lifetime of instructions, steps and resources allocation. We denote by $t_i$ the instructions. An instruction $i$ executed on a stage $j$ is represented by a step $t_{i,j}$. We denote by $t_{i,j,k}$ , the allocation of the resource $k$ by a step $t_{i,j}$. This representation is used in the case of a non-deterministic resource. In case we have a deterministic resource access, we use the notation $t_{i,j,k,occ}$, where *occ* is the accessed occurrence. The right bound and the left bound of the interval $t_{i,j}$ represent the start time and the end time of the step, referred to respectively by $t_{i,j,m}$ and $t_{i,j,M}$.

The problem is defined by:

- the intervals associated with the instruction path,
- the constraints that captures (1) instructions sequence, (2) stages chaining, (3) resources allocation and (4) architecture specific constraints, e.g., out-of-order stages, ...

All these constraints are conjoined to constitute the CSP problem.

We consider the following instruction path to illustrate the constraints. In the first table we represent instructions $i_x$ in assembly code. In the second table, we show the `uses` attribute of the first instruction $i_0$. Each execution step is illustrated (first column) using its representing micro-step $i_{0,y}$.

[3]We use lists instead of sets because two tasks with the same interval and resource must not be mixed.

| | Assembly of the instruction path |
|---|---|
| $i_0$ | `ldr r3, [r11,-#20]` |
| $i_1$ | `cmp r3, #100` |
| $i_2$ | `bgt 838c // end of prefix` |
| $i_3$ | `mov r0, r3, lsl #0` |
| $i_4$ | `sub r13, r11, #12` |
| $i_5$ | `ldmia r13, {, r11, r13, r15}` |

| | Sim-nML "uses" attribute of i0 |
|---|---|
| $i_{0,0}$ | `FE & FBuf,` |
| $i_{0,1}$ | `DE ,` |
| $i_{0,2}$ | `IS & RoB ,` |
| $i_{0,3}$ | `MEM & M.read & R[3].write & R[11].read & RoB,` |
| $i_{0,4}$ | `CM` |

*1) **Instruction constraints**:* We capture here the constraints related to the instructions execution.

When an instruction executes on a stage, it is generally stored in a buffer resource where it remains until it goes to the next stage. The lifetime of the step and the allocation time of the buffer slot are the same.

$$t_{0,0} = t_{0,0,FBuf} \ \wedge \ t_{0,2} = t_{0,2,RoB} \ \wedge \ t_{0,3} = t_{0,3,RoB}$$

During a step, the instruction is contained continuously in a buffer. Otherwise stated, a step yields the control to the following step. Here we present the continuity constraints of the instruction $i0$:

$$t_{0,0} \ \mathbf{Meets} \ t_{0,1} \ \wedge \ t_{0,1} \ \mathbf{Meets} \ t_{0,2} \ \wedge$$
$$t_{0,2} \ \mathbf{Meets} \ t_{0,3} \ \wedge \ t_{0,3} \ \mathbf{Meets} \ t_{0,4}$$

A step is performed if and only if the corresponding stage is available. The start time of the step and the allocation time of the stage are equal.

$$t_{0,0,FE,0} \ \mathbf{Starts} \ t_{0,0} \ \wedge \ t_{0,0,DE,0} \ \mathbf{Starts} \ t_{0,1} \ \wedge$$
$$t_{0,0,IS,0} \ \mathbf{Starts} \ t_{0,2} \ \wedge \ t_{0,3,MEM,0} \ \mathbf{Starts} \ t_{0,3} \ \wedge$$
$$t_{0,0,CM,0} \ \mathbf{Starts} \ t_{0,4}$$

A step $t_{i,j}$ using a register that is produced by another step $t_{i',j'}$ implies that $t_{i',j'}$ must execute before $t_{i,j}$. In our example, $i0$ produces the register `R[3]` on the `ALU` and $i1$ and $i3$ use it.

$$t_{0,3} \ \mathbf{Before} \ t_{1,3} \ \wedge \ t_{0,3} \ \mathbf{Before} \ t_{3,3}$$

*2) **Stage constraints**:* These constraints are drawn from the stages features.

Each step accesses exclusively one stage. If we consider a stage $s$ with $nb\_s$ occurrences, the following cumulative constraint is defined:

$$\mathbf{Cumulative} \ (nb\_s, (t_{i,j,s}, 1), (t_{i\prime,j\prime,s}, 1), \ldots)$$

where $t_{i,j,s}$ are the task set corresponding to the access to the stage $s$. The constraint below concerns the two `ALU` stages of the processor example.

$$\mathbf{Cumulative} \ (2, (t_{1,3,ALU}, 1), (t_{2,3,ALU}, 1),$$
$$(t_{3,3,ALU}, 1), (t_{4,3,ALU}, 1))$$

In-order execution implies that the instructions execute in program order in some stages. In case a stage executes only one instruction in a cycle, every step executes before its successor in the program. Considering a simple-scalar stage $j$, we define the following in-order execution constraint, where $n$ is the instruction path length:

$$\forall i < n.\ t_{i,j}\ \textbf{before}\ t_{i+1,j}$$

The superscalar execution defines a particular execution order for instructions. In a superscalar stage with in-order execution, the steps of two successive instructions may be processed in parallel, *but* the overall program order is maintained. This particular case of temporal relation is not stated in the Allen's intervals algebra. What we need to model is, for two steps $t_{i,j}$ and $t_{i',j'}$, $t_{i,j}.starts \leq t_{i',j'}.starts$. So is defined the temporal relation $StartsBeforeBegin$. Otherwise said, for a stage, of index $j$, with a superscalarity degree $S$, we have :

$$\forall x < S.\ \ t_{i,j}\ \textbf{StartsBeforeBegin}\ t_{i+x,j} \quad (1)$$

In the same context, a stage with a scalarity degree $S$, can execute at most $S$ instructions at the same time. This parallel execution is expressed as a cumulative constraint, where the available resources parameter is the superscalarity degree and the capacity requirement of instructions is set to 1. Let $s$ be a superscalar stage and $t_{i,j,s}$ the steps representing the access to the stage $s$, we consider the following constraint:

$$\textbf{Cumulative}\ (S, \{(t_{i,j,s}, 1)\}) \quad (2)$$

The limited capacity of a superscalar stage requires to force a precedence relation between steps $t_{i,j}$ and $t_{i+S,j}$.

$$t_{i,j}\ \textbf{before}\ t_{i+S,j} \quad (3)$$

The example below shows the constraints related to the commit stage `CM` (2-scalar stage).

(1) $t_{0,4}$ **StartsBeforeBegin** $t_{1,4}$
(1) $t_{1,4}$ **StartsBeforeBegin** $t_{2,4}$ ...
(2) **Cumulative** $(2, (t_{0,4,CM,0}, 1), (t_{1,4,CM,0}, 1),$ ...., $(t_{4,4,CM,0}, 1), (t_{5,4,CM,0}, 1))$
(3) $t_{0,4}$ **Before** $t_{2,4}$
(3) $t_{1,4}$ **Before** $t_{3,4}$...

*3) **Architecture constraints**:* Some architecture features results in dependencies between instructions. For example, a cache line bound may break the parallel execution of instructions because a single cache line can be fetched each cycle. In other words, if two instructions $i$ and $i'$, where $i$ precedes $i'$, are not in the same cache line, they can not be fetched simultaneously. This constraint is generated only for the fetch stage, and is derived from the instruction addresses and the cache line size, specified in the Sim-nML description.

$$t_{i,0}\ \textbf{Before}\ t_{i',0}$$

*4) **Resource constraints**:* Resources used by an instruction in a stage are accessed after the stage is allocated and are released before the instruction leaves the stage. Thus, the live time of the resource allocation is included in the lifetime of the stage allocation. As shown in our example, the instruction $i0$ uses registers `R[3]` and `R[11]` and the memory `M` when executed in the `ALU`.

$$t_{0,3,R,3}\ \textbf{During}\ t_{0,3,MEM,0}\ \wedge$$
$$t_{0,3,R,11}\ \textbf{During}\ t_{0,3,MEM,0}\ \wedge$$
$$t_{0,3,M,0}\ \textbf{During}\ t_{0,3,MEM,0}$$

Buffers are allocated to instructions in a non-deterministic way. Such as stages, a buffer slot can hold, at the same time, only one instruction. A $k$-entry buffer can not hold more than $k$ instructions at a time. We express the buffers exclusive access property as a cumulative constraint, where the resource capacity parameter is the number of buffer slots available and the tasks capacity requirements set to 1. Below is the constraint of the 4-entries fetch buffer `FBuf`.

$$\textbf{Cumulative}\ (4, (t_{0,0,FBuf}, 1), .., (t_{5,0,FBuf}, 1))$$

## V. Experiments

In this section we review the experimentation of our framework (see Figure 1). We also report on the current state of our framework. As stated before, our work flow consists in the following steps:

1) We define the Sim-nML description of the ARMv5 instruction set. Actually, we have extended the existing OTAWA description by the introduced hardware components and the directives related to the execution model, e.g., `uses`.
2) For each target program, we generate instruction paths.
3) Taking a path described trough its instruction addresses, we generate its internal description.
4) From the preceding internal description, we generate the constraints to be satisfied.
5) The generated constraints are solved using the CHOCO solver [2]. We could have also used other solvers implementing the global constraints [5], such as [3].

In Figure 5, we represent the different constraints generated for the example of instruction path considered in the previous section. With respect to the implementation, currently, we reuse the existing processor

descriptions and the OTAWA path analyzer which generates instruction paths from binaries. We have extended the new Sim-nML parser and the internal description generator (section III-E) while we have implemented a sanity check verifier and a constraint generator. In order to resolve the constraints through the CHOCO solver, we generate java code of the constraints. We mention that since our ultimate goal is a certified WCET computation chain, the trace documentation is automatically generated. All this is implemented in the Ocaml language (∼3500 lines).

Our first case study has considered the ARMv5 instruction set and the benchmarks taken from the Mälardalen suite [13]. The Table I concerns the bench *bsort100* and shows computed WCETs with OTAWA and CSP approaches. We give the basic block and the prefix lengths of analysed paths in instruction number. We remark that with respect to the calculated WCET, our results are equals to those of OTAWA, except for paths where :

IS 1-cycle underestimation caused by the lack of a constraint specifying the issue order (actually the program order) of two ready instructions.

SS the *solver strategy* is more precise than that of OTAWA (it is known that the execution graph approach may cause a bit of overestimation in some cases with superscalar microprocessors).

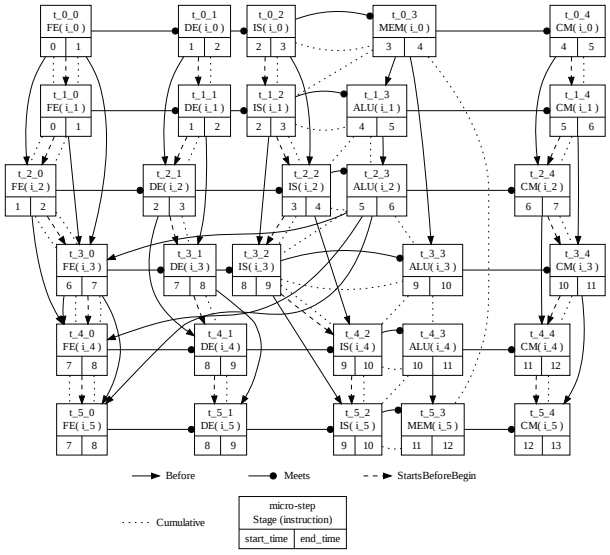


Figure 5: Graph of the instruction path constraints

## VI. Related work

With respect to the model-based approach, several techniques have been used for WCET analysis. Abstract interpretation was used in [10], [26] to analyse models of pipelines and caches. As they remark, faithful abstract models are hard to obtain. Processors are described at the circuit level through a VHDL description [25]. Although, the complexity of this low level description entails a lack of accuracy in the WCET analysis, efficient tools supporting this approach become nowadays available [1]. Another formal approach has been investigated by [7], [8] by modeling the processor and the program using timed automata. Verification and WCET analysis are achieved by model-checking but the time cost and the combinatory explosion of the state space of such a method limits the capacity of the approach to handle complex architectures. Other works are based on ADL [21] for architecture modeling. ADL are classified into categories. Behavior-centric ADLs like ISDL [14] and nML [9] describes the processor by its *instruction set architecture*. Architecture-centric languages like MIMOLA [16] capture the *structure of the processor*. Mixed languages that bring together the two processor aspects have also been proposed [22], [28]. Thanks to high level processor descriptions, it becomes possible to make high level analysis of processor behaviors. Another interesting feature of high level descriptions is that they enable early verification [20].

Table I: WCETs of bench Bsort100.

| Basic block length | Prefix length | OTAWA WCET | CSP WCET | Difference | |
|---|---|---|---|---|---|
| 8 | 0 | 7 | 8 | -1 | IS |
| 15 | 6 | 12 | 11 | 1 | SS |
| 5 | 4 | 5 | 3 | 2 | SS |
| 5 | 3 | 5 | 5 | 0 | |
| 7 | 3 | 6 | 3 | 3 | SS |
| 5 | 3 | 6 | 6 | 0 | |
| 3 | 5 | 1 | 0 | 1 | SS |
| 16 | 5 | 12 | 12 | 0 | |
| 3 | 3 | 3 | 0 | 3 | SS |
| 6 | 3 | 6 | 6 | 0 | |
| 5 | 1 | 6 | 6 | 0 | |
| 30 | 14 | 16 | 17 | -1 | IS |
| 6 | 14 | 6 | 6 | 0 | |
| 1 | 5 | 5 | 5 | 0 | |
| 5 | 4 | 6 | 5 | 1 | SS |
| 6 | 28 | 3 | 2 | 1 | SS |
| 11 | 2 | 9 | 9 | 0 | |
| 5 | 4 | 6 | 5 | 1 | SS |
| 5 | 13 | 3 | 2 | 1 | SS |
| 18 | 3 | 13 | 11 | 2 | SS |
| 5 | 3 | 6 | 6 | 0 | |
| 5 | 16 | 6 | 3 | 3 | SS |
| 4 | 3 | 4 | 4 | 0 | |
| 5 | 9 | 3 | 3 | 0 | |
| 6 | 3 | 5 | 2 | 3 | SS |
| 5 | 3 | 5 | 5 | 0 | |

## VII. Conclusion

In this paper, we have presented an ADL-based approach for WCET analysis. To achieve this goal, we have proposed an extension to the Sim-nML language.

Using the description of processors and instructions sets, we generate a constraint based time analysis. This constraint description is solved using a CSP solver. To the best of our knowledge, WCET analysis through a constraint based declarative approach is new. Currently, we have made a small number of experiments. It should be stressed that the speed of computation is close to that of OTAWA. However, currently, we have not made any optimization, e.g., variable range strengthening, over the generated constraints. We hope that more refined constraints will provide better performance. With respect, to the architecture features taken into account, we believe that the constraint approach is more modular than current approaches [17], [24] and consequently can be extended easily. As a matter of fact, for our future work we intend to take into account cache analysis, bus access [12], [19] and to stress the proposed approach with exotic actual microprocessor implementations. Moreover, as said in the beginning, since our ultimate goal is a certified wcet computation, such a modular approach should make the validation work easier.

**Aknowledgement.** We would like to thank Marie de Roquemaurel for introducing us to the world of Constraint Satisfaction Problems.

## REFERENCES

[1] aiT Worst-Case Execution Time Analyzers. http://www.absint.com/ait/.

[2] Choco: an Open Source Java Constraint Programming Library. http://choco.mines-nantes.fr.

[3] Gecode: GEneric COnstraint Development Environment. http://www.gecode.org, 2006.

[4] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

[5] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global Constraint Catalog. http://www.emn.fr/z-info/sdemasse/gccatold/index.html, 2005.

[6] A. Burns. *Programming in Occam 2*. Addison-Wesley, 1988.

[7] F. Cassez. Timed games for computing wcet for pipelined processors with caches. In *International Conference on Application of Concurrency to System Design (ACSD*, 2011.

[8] A. Dalsgaard, M. Olesen, M. Toft, R. Hansen, and K. Larsen. METAMOC: Modular execution time analysis using model checking. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.

[9] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. European Design and Test Conference (EDTC), 1995.

[10] C. Ferdinand and R. Wilhelm. Fast and efficient cache behavior prediction. Technical report, Universitäts und Landesbibliothek, 1997.

[11] M. Freericks. The nml machine description formalism. Technical Report 1991/15, TU Berlin, 1991.

[12] N. Guan, M. Lv, and W. Y. 0001. Fifo cache analysis for wcet estimation: a quantitative approach. In *Design, Automation and Test in Europe (DATE)*, 2013.

[13] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.

[14] G. Hadjiyiannis, S. Hanono, and S. Devadas. Isdl: an instruction set description language for retargetability. Design Automation Conference (DAC), 1997.

[15] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[16] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. design automation for embedded systems. In *In Design Automation for Embedded Systems*, 1998.

[17] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 2006.

[18] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Design Automation Conference (DAC)*, 1995.

[19] M. Lv, N. Guan, Q. Deng, G. Yu, and W. Y. 0001. Mcait - a timing analyzer for multicore real-time software. In *International conference on Automated technology for verification and analysis (ATVA)*, 2011.

[20] P. Mishra and N. Dutt. Modeling and validation of pipeline specifications. *ACM Trans. Embedded Computer Systesms*, 2004.

[21] P. Mishra and N. Dutt. *Processor description languages: applications and methodologies*, chapter 2. Morgan Kaufmann Publishers/Elsevier, 2008.

[22] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, 2000.

[23] T. Ratsiambahotra, H. Cassé, and P. Sainrat. A versatile generator of instruction set simulators and disassemblers. In *Proceedings of the 12th international conference on Symposium on Performance Evaluation of Computer & Telecommunication Systems (SPECTS)*, 2009.

[24] C. Rochange and P. Sainrat. A context-parameterized model for static analysis of execution times. *Transactions on High-Performance Embedded Architectures and Compilers II*, 2009.

[25] M. Schlickling and M. Pister. A framework for static analysis of VHDL code. In *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[26] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. Languages, Compilers, and Tools for Embedded Systems (LCTES), 1999.

[27] E. T. *Foundations of constraint satisfaction (Computation in Cognitive Science)*. Computation in cognitive science. Academic Press, 1993.

[28] C. Tradowsky, F. Thoma, M. Hübner, and J. Becker. Lisparc: Using an architecture description language approach for modelling an adaptive processor microarchitecture. In *International Symposium on Industrial Embedded Systems (SIES)*, 2012.

[29] R. J. Vanderbei. Linear programming: Foundations and extensions, 1996.

[30] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.